Lecture 15
Rotations in 3D

Rotations

What is a rotation anyway?

It is a **rigid** transformation

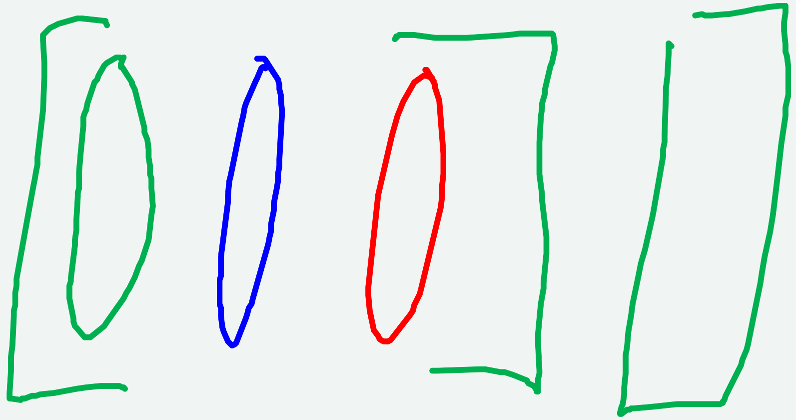
- preserves **distances**
- preserves **handedness**

Two type of rigid transformations

- translations (all points change the same amount)
- rotations (one point - the **center** - doesn't change)

Rotations are Linear Transformations

Orthonormal Matrices



- all rows (columns) have unit length
- all rows (columns) mutually orthogonal

Positive determinant (preserve handedness)

- Not all matrices are rotations

ortho-normal
positive

Rotation Facts

- Have an **inverse**
- The inverse is the transpose (only rotations)

transp

- Closed set (composition yields another rotation)
- Associates (do operation in any order)
- Does not commute (in general)

$R_1 R_2 \neq R_2 R_1$

$R_1 R_2 R_3 R_4 = R_*$

$(R_1 R_2) R_3 R_4$

$R_2 (R_3 R_4)$

There is a **center** and **axis** of rotation

Rotation vs. Orientation

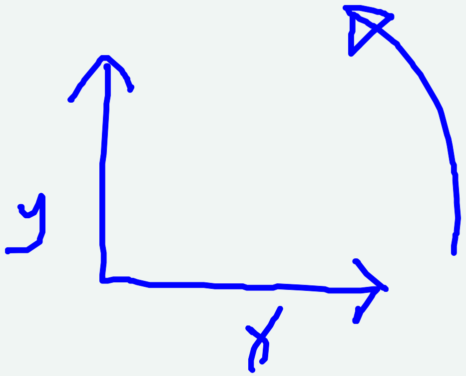
transformation

state

2D Rotations in 3D

Center of rotation is an axis

Normal 2D rotation is about the Z axis



Center of Rotation

Rotation is about the origin

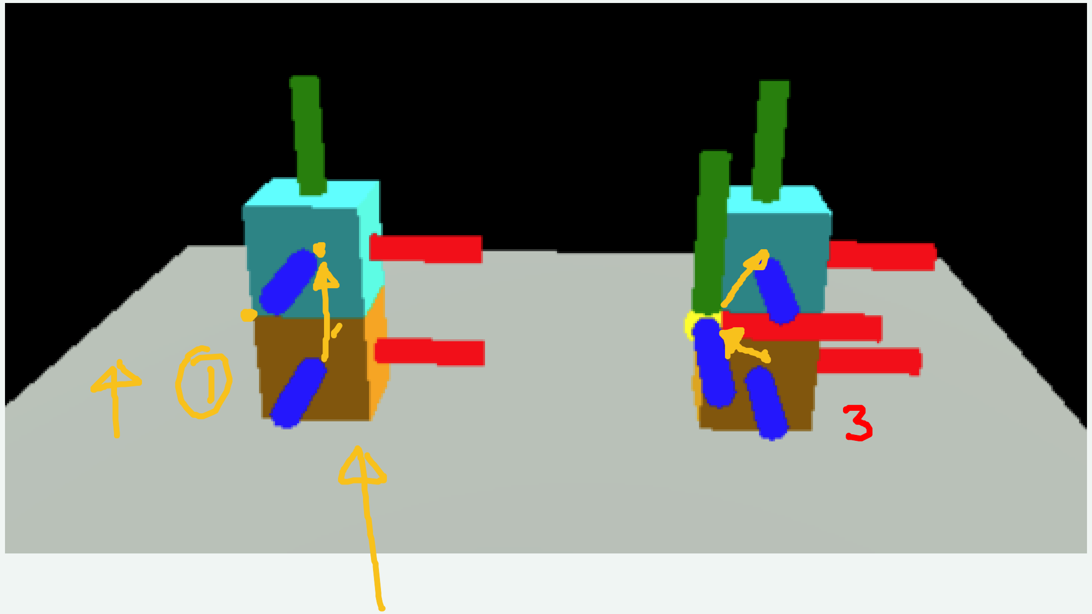
Shift the origin to where you want to rotate

Shift it back after rotation

Center of Rotation in THREE

Use a `Group` to put the center in the right place

Put the object in the group (relative to object's center)



cube 1

y+1

cube 2

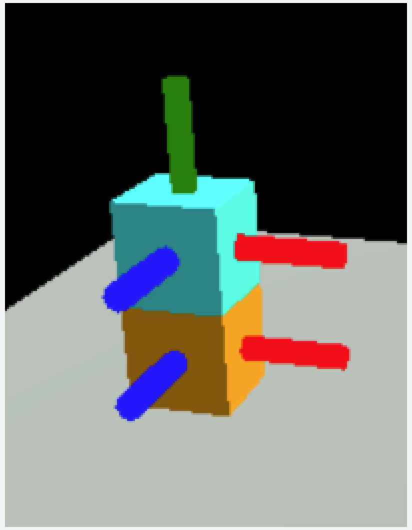
cube3

group 1

cube 4

stack demo

Default Hierarchy



Cubes have their center in the center

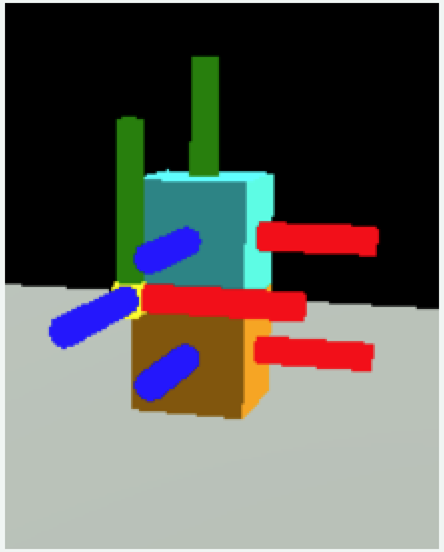
Notice the distance to stack

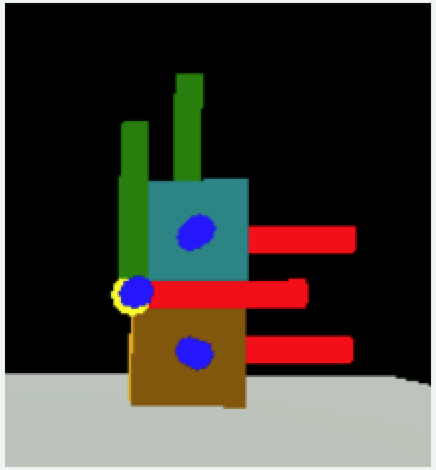
Place one cube in another

```
let cube1 = cube("orange");
cube1.position.set(-2,.5,3);
let cube2 = cube("cyan");
cube1.add(cube2);
cube2.position.set(0,1,0);
scene.add(cube1);
```

```
function cube(color) {
    return new T.Mesh(new T.BoxGeometry(),
                      new T.MeshStandardMaterial({color:color}));
}
```

With a Group





Place the group at the corner of cube

Place the 2nd cube in group

Rotate the Group

```
let cube3 = cube("orange");
cube3.position.set( 2,.5,3);
let group1 = new T.Group();
cube3.add(group1);
group1.position.set(-.5,.5,.5);
let cube4 = cube("cyan");
group1.add(cube4);
cube4.position.set(.5,.5,-.5);
scene.add(cube3);
```

Rotations about Axes

Rotation about X

Rotation about Y

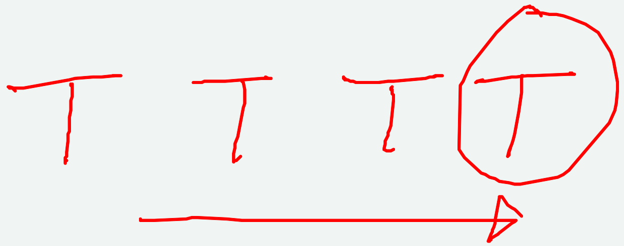
Rotation about Z

demo: EulerToy 1

Axes in the world vs. local axes

demo: EulerToy 1

global



local

Sequences of Rotations

demo: Euler Toy 2

Euler's Theorems

1. Any rotation can be represented as a single rotation about some axis
2. Any rotation can be represented as a sequence of three rotations about a fixed axis

each

X Y Z

Euler Angles

X Z X

Axis Angle

~~X X Z~~

Euler Angles

Be careful: Euler invented many different kinds of angles

Z X Y

global local

local

Rotation around 3 fixed axes

- Could be any order (XYZ, ZYX, ZXY)
- Can repeat (ZXZ)
- Can be local or global
  - easier to think "global to the left"

Z

Play with them

Demo: EulerToy 3

More on Euler Angles

Earlier rotations change the meaning of later ones

Order matters

Local to global (or global to local)

(demo)

(incremental rotations in the workbook)

Composing Rotations

In a single axis (like in 2D):

$$R_z(a) \circ R_z(b) = R_z(a+b)$$

With different axes, this does not hold!

$$R_x(a) \circ R_y(b) = R_?(?)$$

And things in between cause problems

$$R_x(a) \circ R_y(b) \circ R_x(c) \neq R_x(a+c)R_y(b)$$

Getting Stuck

Rotate about X then Y

Rotate about Z is the same as the first rotate about X

Gimbal Lock

No matter what X is, Y=90 aligns Z with it

- There is no way to get the Y axis out of the X=0 plane
- We lost a degree of freedom

(demo EulerToy3)

Two ways to the same place

Rotate about X then Y

Rotate about Y then Z

same! (but different path)

(90,90,0) = (0,90,90) - but can't interpolate!

(demo EulerToy4)

Euler Angles (XYZ)

Good:

Easy for 1 axis

Easy for simple combinations

Bad:

Hard to get what you want (unintuitive combinations)

Can't interpolate

Gimbal lock (can't get there from here)

Axis Angle (Euler's other theorem)

Demo: et-axisangle

Axis Angle

Downsides:

- hard to figure out what axis
- hard to compose

Rotation Vector

Store the angle as the magnitude of the axis

$x \quad y \quad z$ $\qquad \theta$

$\theta\hat{x} \quad \theta\hat{y} \quad \theta\hat{z}$

Rotation Matrices

- hard to interpret
- easy to "drift"
- hard to insure it's a rotation
  - Gramm Schmidt Orthonomalization

$\begin{bmatrix} & & \\ & & \\ & & \end{bmatrix}$ 9 numbers!

easy to compose

easy to apply to a point

Unit Quaternions

$\| a \quad b \quad c \quad d \| = 1$

4 numbers:

- Axis angle: $\theta, \hat{\mathbf{n}}$ $\Rightarrow$ into 4 numbers
- Unit quaternion: $\cos(\frac{\theta}{2}), \underbrace{\sin(\frac{\theta}{2})}_{1}\hat{\mathbf{n}}$
- Will have magnitude 1

Why?

What is a Quaternion anyway?

4-dimensional complex number

Consider 2D complex numbers $(a + b\mathbf{i})$

- we can do arithmetic on them
- multiplication is meaningful

4D Complex Numbers?

Don't worry... you can look up:

- formulas to multiply
- formulas to convert to Matrix form
- formulas to interpolate (and preserve unit-ness)

$w \quad x \quad y \quad z$

real $\quad$ imaginary

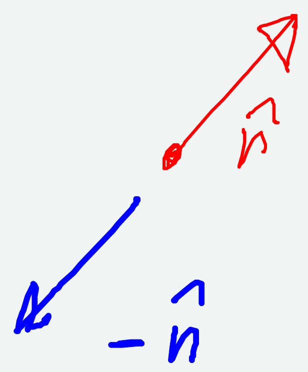
$\cos\frac{\theta}{2} \quad \sin\frac{\theta}{2}\,\hat{n}$

but you should know...

- these formulas exist
- multiplication preserves unit-ness
- multiplication composes transformations

Why is this better? (or is it?)

- No Gimbal Lock (but antipodes)
- Represents orientations
- Close things are close (except for sign flips)



But Really:

- Easy to compose
- Easy to interpolate (not linear interpolation)
- Other nice math (interpolation)
- 3x3 rotation matrices are a pain
- Easy to fix drift

Convert to Quaternions

(Other direction is MUCH harder)

Axis angle $(\theta, \hat{\mathbf{v}})$ -> $(\cos(\frac{\theta}{2}), sin(\frac{\theta}{2})\hat{\mathbf{v}}))$

Euler Angles XYZ (x,y,z)

- make a quaterion for each $(\cos(\frac{x}{2}), sin(\frac{x}{2}[1,0,0])))$
- multiply the quaternions together

THREE.js and rotations

Internally, stores quaternions

- it provides all conversions
- it does conversions automatically (beware errors!)
- it provides good quaternion functions
- it gives you operations using other forms
    - axis angle, euler angle, matrix,

You never **need** to see the quaternions... unless you want to

THREE and Rotations

State set what current coordinate sys is

(variables / orientation)

matrix (normalMatrix, ...)

position

scale

quaternion

rotation → euler

Transforms apply in current

(motions / rotations)

applyMatrix4

translate (x,y,z, onAxis, ...)

applyQuaternion

rotate (x,y,z, onAxis, ...)

`lookAt`, `setFrom` are special (a method that sets) an absolute orientation

Internally...

The **quaternion** is used for everything

If you do something else, it is converted to the quaternion

If you apply a matrix it must be **decomposed** into rotate, translate, scale

```
applyMatrix: function ( matrix ) {
        this.matrix.multiplyMatrices( matrix, this.matrix );
        this.matrix.decompose( this.position, this.quaternion, this.scale );
}, // in Object3D.js
```

Internally

```
translateX: function () {
        var v1 = new Vector3( 1, 0, 0 );
        return function translateX( distance ) {
                return this.translateOnAxis( v1, distance );
        };
}(),
translateOnAxis: function () {
        // translate object by distance along axis in object space
        // axis is assumed to be normalized
        var v1 = new Vector3();
        return function translateOnAxis( axis, distance ) {
                v1.copy( axis ).applyQuaternion( this.quaternion );
                this.position.add( v1.multiplyScalar( distance ) );
                return this;
        };
}(),
```

Old School JavaScript hidden constant

```
translateX: function () {
        var v1 = new Vector3( 1, 0, 0 );
        return function translateX( distance ) {
                return this.translateOnAxis( v1, distance );
        };
}(),
```

A Special Rotation: LookAt

Point the Z axis towards a point

- Useful for cameras
- Useful for other objects

Note this is not unique

- Only specifies 2 dergees of freedom

Up Vector!

Lookfrom / Lookat / Up

- In Three
  - `position` of object center
  - `lookat` method
  - `up` vector (object property)

Internally, it will convert to quaternion

Geometric Derivation

1. Point z at target

   normalize(at - from) $\widehat{(\text{at} - \text{from})}$

2. Find x (right) as $\widehat{up} \times z$
3. Find y (local up) as $z \times x$

Notice: we have built a rotation matrix!

It has all the right properties

We never figured out angles

Rotations Summary: What you need to know

1. Basic facts (rigid, orthonormal, composition, ...)
2. Single Axis Rotations
3. Euler Angles - be able to think about them
    - local vs. global
    - how things compose (and complexities)
4. Axis Angle forms - understand what they are
5. Quaternions
    - basic facts - and know they are inside THREE
6. Lookfrom/Lookat/VUp
7. Use in THREE (including centers)